Resource-Bound Quantification for Graph Transformation

Paolo Torrini
University of Leicester
pt95@mcs.le.ac.uk

Reiko Heckel
University of Leicester
reiko@mcs.le.ac.uk

Graph transformation has been used to model concurrent systems in software engineering, as well as in biochemistry and life sciences. The application of a transformation rule can be characterised algebraically as construction of a double-pushout (DPO) diagram in the category of graphs. We show how intuitionistic linear logic can be extended with resource-bound quantification, allowing for an implicit handling of the DPO conditions, and how resource logic can be used to reason about graph transformation systems.

1 Introduction

Graph transformation (GT) combines the idea of graphs, as a universal modelling paradigm, with a rule-based approach to specify the evolution of systems. It can be regarded as a generalisation of term rewriting. Among the several formalisations of GT based on algebraic methods, the double-pushout approach (DPO) is one of the most influential [12]. Intuitionistic linear logic (ILL) has been applied to the representation of concurrent systems [5, 1, 15], in relationship with Petri nets, multiset rewriting and process calculi. This paper reports work on the embedding of DPO-GT into a variant of quantified intuitionistic linear logic with proof terms (HILL). The general goal is to build a bridge between constructive logic and the specification of concurrent systems based on graph transformation — with special attention to model-driven software development. Representing model-based specifications of object-oriented programs as proof terms could be useful for mechanised verification.

Hypergraphs are a generalisation of graphs allowing for edges that connect more than two nodes (hyperedges). Term-based algebraic presentations of DPO-GT usually rely on hypergraphs and hyperedge replacement [7]. Intuitively, an hypergraph can be defined in terms of parallel compositions of components — where a component can be either the empty hypergraph, a node, or an edge component (an hyperedge with attached nodes). A transformation may delete, create or preserve components.

It can be convenient to represent an hypergraph as a logic formula, where hyperedges are predicates ranging over nodes, and composition is represented by a logic operator. There are naming aspects that need to be addressed in representing transformation. In particular, (1) renaming is needed in order to reason about models up to isomorphism, and (2) the representation of transformation rules involves abstraction from component names. Transformation cannot be represented directly in terms of either classical or intuitionistic consequence relation, because of weakening and contraction. Accounts based on hyperedge replacement [7] and second-order monadic logic with higher-order constructors [8] rely on extra-logical notions of transformation. Substructural logics offer a comparatively direct way to express composition as multiplicative conjunction ($\otimes$), and transformation in terms of consequence relation, with associated linear implication ($\multimap$). This is the case with linear logic [5, 9, 6] as well as with separation logic [10].

There are further semantic aspects to be considered. One is the double status of nodes. From the point of view of transformation, each node as graph component is a linear resource. From the point of

M. Florido and I. Mackie (Eds.): First International
Workshop on Linearity (LINEARITY 2009)
EPTCS 22, 2010, pp. 14–25, doi:10.4204/EPTCS.22.2

view of the spatial structure, a node represents a connection between edge components — therefore it is a name that may occur arbitrarily many times. Another aspect is the asymmetry between nodes and edges with respect to deletion. An edge can be deleted without affecting the nodes, whereas it makes little sense to delete a node without deleting the edges it is attached to. On the other hand, by default, edge deletion should not trigger node deletion. There are systems in which isolated nodes are disregarded, but this is not generally the case when dealing with hierarchical graphs [3, 11, 14], especially in case nodes represent subgraphs.

We focus on the problem of representing at the object level a constructive notion of renaming, which behaves injectively, unlike instantiation of quantified variables and substitution of meta-variables. Here we rely on a representation of names as terms that refer to locations, relying on the linear aspect of the logic, and extending the operational approach presented in [22]. Our goal is more specific than that of higher level approaches to names with binding based on nominal logic [13, 21]. In section 2 we provide a categorical presentation of typed DPO-GT, independently of syntactical formalisation. In section 3 we present a form of linear lambda calculus with dependent types, extended with a notion of location (with $\downarrow$), and a resource bound quantifier $\hat{\exists}$ to represents name hiding. In section 4 we show how GT systems can be embedded in HILL.

1.1 Overview

By extending ILL with quantification one can hope to deal with abstraction, and therefore to reason about GT systems in logic terms up to α-renaming. However, this requires coping with the difference between variables and names. As a simple example, consider a graph given by an r-typed edge $r(x,y)$ that connects two distinct nodes x,y, and a rule that replaces the r-typed edge with a b-typed one, i.e. $r(n_1,n_2)$ with $b(n_1,n_2)$. In order to abstract from node names, assuming Q_1,Q_2 are quantifiers, we need to introduce an abstract representation $Q_1xy.r(x,y)$ for the graph. Intuitively, we could choose between (1) $(Q_1xy.r(x,y)) \multimap (Q_1xy.b(x,y))$ and (2) $Q_2xy.r(x,y) \multimap b(x,y)$ to represent the rule. It is not difficult to see that no interpretation of Q_1,Q_2 in terms of $\exists,\forall$ is completely satisfactory. $\exists xy.b(x,y)$ follows from $b(n_1,n_1)$, and $\forall xy.r(x,y)$ implies $r(n_1,n_1)$. In general, neither existential nor universal quantification can prevent the identification of distinct variables through instantiation with the same term — i.e. they do not behave injectively with respect to multiple instantiation.

Freshness quantification ($\mathcal{N}$), associated to name restriction in the context of MF-logic [13, 19], relies on a notion of bindable atom to represent names, an account of substitution in terms of permutation and of α-equivalence in terms of equivariance. A typing for restriction can be found in [21]. However, with standard quantifiers, as well as with freshness, one has that $\exists x.\alpha$, $\forall x.\alpha$, $\mathcal{N}x.\alpha$ are logically equivalent to α whenever x does not occur in α — we can call this property η-congruence.

In this paper, we define a quantifier ($\hat{\exists}$) that keeps the above-mentioned graph-specific aspects into account — in particular, it behaves injectively, and it satisfies the algebraic properties of name restriction except for η-congruence. $\hat{\exists}$ has a separating character (though in a different sense from the intensional quantifiers in [20]), by implicitly associating each bound variable to a linear resource. It has a freshness character in requiring the relationship between witness terms and bound variables to be one-to-one — this makes the introduction rules of $\hat{\exists}$ essentially invertible, unlike standard existential quantification.

$\hat{\exists}$ can be understood operationally by saying that, with its introduction, given an instance $M :: \alpha[D/x]$, all the occurrences of the non-linear term D (the witness) in the instance become hidden, and in a sense the witness becomes linear. In $\hat{\varepsilon}(D|n).M :: \hat{\exists}x.\alpha$, the witness may still occur in the term, but it has been exhaustively replaced with a bound variable in the type, and it has become associated with the linear location n. We rely on a meta-level representation of hiding in terms of existential quantification, as

usually found in dependent type theory. The difference lies with the exhaustive character (a freshness condition) and with the injective association to linear resources. In this paper we stop short of introducing restriction ν at the object language level. This could be done, by using as interpretation for $\hat{\exists}$ terms such as $\nu x.n_D \otimes M[x/D]$. However, extending lambda-calculus with restriction involves more than technicalities — see [18, 21]. Here we limit ourselves to consider hiding, by using terms such as $\hat{\varepsilon}(D|n).M = n \otimes D \otimes M$, with D and n both hidden by the type.

Non-linear terms can be contracted — i.e. two of the same type can be merged. This can explain multiple occurrences of a term in an expression, assuming the point of view of linearity as default. Technically, the approach we use for names consists of associating the naming term D to a location, in order to prevent contraction for the free variables in D (the nominal variables), hence for D itself, thus closing their scope. Assuming linearity for locations, η-equivalence fails on one hand, and on the other the set of names turns out minimal — unlike in [21], where the name space is affine.

2 Hypergraphs and their transformations

A hypergraph (V,E,s) consists of a set V of vertices, a set E of hyperedges and a function $\mathsf{s} : E \to V^*$ assigning each edge a sequence of vertices in V. A morphism of hypergraphs is a pair of functions $\phi_V : V_1 \to V_2$ and $\phi_E : E_1 \to E_2$ that preserve the assignments of nodes — that is, $\phi_V^* \circ \mathsf{s}_1 = \mathsf{s}_2 \circ \phi_E$. By fixing a type hypergraph $TG = (\mathscr{V}, \mathscr{E}, \mathsf{ar})$, we are establishing sets of node types $\mathscr{V}$ and edge types $\mathscr{E}$ as well as defining the arity $\mathsf{ar}(a)$ of each edge type $a \in \mathscr{E}$ as a sequence of node types. A TG-typed hypergraph is a pair $(HG, type)$ of a hypergraph HG and a morphism $type : HG \to TG$. A TG-typed hypergraph morphism $f : (HG_1, type_1) \to (HG_2, type_2)$ is a hypergraph morphism $f : HG_1 \to HG_2$ such that $type_2 \circ f = type_1$.

A *graph transformation rule* is a span of injective hypergraph morphisms $L \xleftarrow{l} K \xrightarrow{r} R$, called a *rule span*. A hypergraph transformation system (GTS) $\mathscr{G} = \langle TG, P, \pi, G_0 \rangle$ consists of a type hypergraph TG, a set P of rule names, a function mapping each rule name p to a rule span $\pi(p)$, and an initial TG-typed hypergraph G_0. A *direct transformation* $G \xRightarrow{p,m} H$ is given by a *double-pushout (DPO) diagram* as shown below, where (1), (2) are pushouts and top and bottom are rule spans. For a GTS $\mathscr{G} = \langle TG, P, \pi, G_0 \rangle$, a derivation $G_0 \Longrightarrow G_n$ in $\mathscr{G}$ is a sequence of direct transformations $G_0 \xRightarrow{r_1} G_1 \xRightarrow{r_2} \cdots \xRightarrow{r_n} G_n$ using the rules in $\mathscr{G}$. An hypergraph G is *reachable* in $\mathscr{G}$ iff there is a a derivation of G from G_0.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle d} & (2) & \downarrow{\scriptstyle m^*} \\ G & \xleftarrow[g]{} & D & \xrightarrow[h]{} & H \end{array}$$

Intuitively, the left-hand side L contains the structures that must be present for an application of the rule, the right-hand side R those that are present afterwards, and the gluing graph K (the *rule interface*) specifies the "gluing items", i.e., the objects which are read during application, but are not consumed. Operationally speaking, the transformation is performed in two steps. First, we delete all the elements in G that are in the image of $L \setminus l(K)$ leading to the left-hand side pushout (1) and the intermediate graph D. Then, a copy of $L \setminus l(K)$ is added to D, leading to the derived graph H via the pushout (2). The first step (deletion) is only defined if the built-in application condition, the so-called gluing condition, is satisfied by the match m. This condition, which characterises the existence of pushout (1) above, is usually presented in two parts.

Identification condition: Elements of L that are meant to be deleted are not shared with any other

elements — i.e., for all $x \in L \setminus l(K)$, $y \in L$, $m(x) = m(y)$ implies $x = y$.

Dangling condition: Nodes that are to be deleted must not be connected to edges in G that are not to be deleted — i.e., for all $v \in G_V$, for all $e \in G_E$ such that v occurs in $\mathsf{s}(e)$, then $e \in m_E(L_E)$.

The first condition guarantees two intuitively separate properties: first — nodes and edges that are deleted by the rule are treated linearly, i.e., m is injective on $L \setminus l(K)$; second — there must not be conflicts between deletion and preservation, i.e., $m(L \setminus l(K))$ and $m(l(K)$ are disjoint. The second condition ensures that after the deletion action, the remaining structure is still a graph, and therefore does not contain edges short of a node.

As terms are often considered up to renaming of variables, it is common to abstract from the identity of nodes and hyperedges considering hypergraphs up to isomorphism. However, in order to be able to compose graphs by gluing them along common nodes, these have to be identifiable. Such potential gluing points are therefore kept as the *interface* of a hypergraph, a set of nodes I (external nodes) embedded into HG by a morphism $i : I \to HG$. An abstract hypergraph $i : I \to [HG]$ is then given by the isomorphism class $\{i' : I \to HG' \mid \exists \text{ isomorphism } j : HG \to HG' \text{ such that } j \circ i = i'\}$.

If we restrict ourselves to rules with interfaces that are *discrete* (i.e., containing only nodes, but no edges), a rule can be represented as a pair of hypergraphs with a shared interface I, i.e., $\Lambda I.L \Longrightarrow R$, such that the set of nodes I is a subgraph of both L, R. This restriction does not affect expressivity in describing individual transformations because edges can be deleted and recreated, but it reduces the level concurrency. In particular, concurrent transformation steps can no longer share edges because only items that are preserved can be accessed concurrently.

Syntactical presentations of GT based on this semantics have been given, relying on languages with a monoidal operator, a name restriction operator and an appropriate notion of rule and matching [7].

3 Linear lambda-calculus

We give a constructive presentation of an extension of intuitionistic linear logic based on sequent calculus, using a labelling of logic formulas that amounts to a form of linear λ-calculus [1, 2, 4, 17]. We build on top of a system with ILL propositional type constructors $\multimap, \otimes, \mathbf{1}, !$ and universal quantifier $\forall$ (we omit $\to$ as case of the latter). Each of these can be associated to a λ-calculus operator [1, 17]. Linear implication ($\multimap$) is used to type linear functions, and we use $\hat{\lambda}$ for linear abstraction (with $\hat{}$ for linear application), to distinguish it from non-linear λ (typed by $\forall$). We assume α-renaming and β-congruence for λ and $\hat{\lambda}$ (with linearity check for the latter). The operator associated to $\otimes$ is parallel composition, with nil as identity. The $!$ is interpreted as closure operator. We extend this system with a dependant type constructor $\lfloor$ to introduce a notion of naming, and with a resource-bound existential quantifier $\hat{\exists}$ associated with linear hiding.

We rely on a presentation based on double-entry sequents [16, 17]. A sequent has form $\Gamma; \Delta \vdash N :: \alpha$, where Δ is the linear context, as list of typed linear variables $(v, u, \ldots)$ among which we distinguish location variables $(n, m, \ldots)$, and Γ is the non-linear context, as list of typed variables $(x, y, \ldots)$. We implicitly assume permutation and associativity for each context, and use a dot $(\cdot)$ for the empty one. $N :: \alpha$ is a typing expression (typed term) where N is a label (term) and α is a logic formula (type). $\vdash$ represents logic consequence, whenever we forget about terms. We need to keep track of the free nominal variables in order to constrain context-merging rules, and to this purpose we annotate each sequent with the set Σ of such variables, writing $[\Sigma]; \Gamma; \Delta \vdash N :: \alpha$, where Σ is a subset of the variables declared in Γ.

Derivable sequents are inductively defined from the axioms and proof rules in section 3.1, and with them the sets of well-formed terms and non-empty types. Notice that the definition of derivation includes

that of the free nominal variable set Σ.

Syntactically, terms are $M = v \mid x \mid n \mid \mathsf{nil} \mid N_1 \otimes N_2 \mid \hat{\varepsilon}(D|N).N \mid \lambda x.N \mid \hat{\lambda} u.N \mid N_1 \hat{} N_2 \mid ND \mid !N \mid \mathsf{discard}\ \Gamma\ \mathsf{in}\ N \mid \mathsf{copy}(x)$, where non-linear terms (those that do not contain free linear variables) are $D = x \mid !N$. Formulas (or types) are $\alpha = A \mid E(D_1 : \alpha_1, \ldots, D_n : \alpha_n) \mid \mathbf{1} \mid \alpha_1 \otimes \alpha_2 \mid \alpha_1 \multimap \alpha_2 \mid !\alpha_1 \mid \forall x : \beta.\alpha \mid \hat{\exists} x : \beta.\alpha \mid \alpha \lfloor D$. Linear equivalence is defined by $\alpha \hat{=} \beta \ =_{df}\ (\alpha \multimap \beta) \otimes (\beta \multimap \alpha)$. Patterns are terms given by $P = v \mid x \mid n \mid \mathsf{nil} \mid P_1 \otimes P_2 \mid \hat{\varepsilon}(x|n).P \mid !P \mid \mathsf{copy}(x)$. They are used in *let* expressions, defined as $\mathsf{let}\ P = N_1\ \mathsf{in}\ N_2 \ =_{df}\ N_2[N_1/P]$.

We say that γ is an *atomic type* whenever either $\gamma = A_i$ or $\gamma = E_i(D_1 : \alpha_1, \ldots, D_k : \alpha_k)$ where $\alpha_1, \ldots, \alpha_k$ are closed types. We take $A_0, A_1, \ldots$ to be atomic closed types, meant to represent GT node types. A node of type A is represented as non-linear variable of type $!A$ (see section 4). We take $E_0, E_1, \ldots$ to stand for atomic type constructors, meant to be associated with GT edge types. A HILL type $E(D_1 : T_1, \ldots, D_k : T_k)$ (by annotating terms with their types) is meant to represent a GT edge type $E(A_1, \ldots A_k)$, if we forget node terms, whenever $T_1 = !A_1, \ldots T_k = !A_k$.

Semantically, we assume that $v \in LV$, a set of linear variables, $x \in UV$, a set of non-linear variables, and $n \in LOC \subset LV$, a set of linear variables that evaluate to themselves and that we call locations. Given a derivable sequent Ω, the non-linear context Γ can be interpreted as a partial function $UV \to TY$ such that $\Gamma(x)$ is either closed or undefined for each x, and the linear context Δ as a partial function $LV \to TY$, such that for each $n \in LOC$, if defined $\Delta(n)$ has form $\alpha \lfloor D$ (*location type*) with α closed, and D non-linear term of type α. The free variables in Ω are those for which either Γ or Δ is defined. $FV_\Omega(N)$ denotes the free variables occurring in N, $FV_\Omega(\alpha)$ those occurring in α (subscripts omitted in case of no ambiguity). We require for $\Delta_{|LOC}$ (restriction of Δ to LOC) to satisfy the following *separation condition*: for each $n, m \in LOC, n \neq m$ if defined $FV(\Delta_{|LOC}(n)) \cap FV(\Delta_{|LOC}(m)) = \emptyset$. We say that a location is *proper* if $FV(\Delta_{|LOC}(n)) \neq \emptyset$, *improper* otherwise.

The location typing assignment $n : \beta \lfloor D$ says that D of type β is the *naming term* of n, that n is the location (β-location) of D, and that the variables that occur free in D (*nominal variables*) are located at n. We denote by $Names_\Omega$ the subset of well-typed terms that occur as naming terms in Ω. We use $FN(D)$ (resp. $FN(\alpha)$) to denote the nominal variables that occur free in D (resp. α), and we denote by Σ the set of the free nominal variables in Ω, i.e. the free variables that occur in $Names_\Omega$. Variables become nominal when located. Semantically, a name can be thought of as a pair (D, n) (naming term and location). The separation condition implies that Δ_{LOC} is injective in a strong sense — different locations are associated with names that do not share free variables.

The separation condition required by the definition of $\Delta_{|LOC}$ needs to be enforced explicitly, in all the context-merging rules. In order to express the constraint, we annotate sequents with the recursively computed set Σ (in brackets) of the free naming variables. We take $[\Sigma, \Sigma', x]$ to represent the disjoint union $\Sigma \uplus \Sigma' \uplus \{x\}$, and $[\Sigma - x]$ to represent $\Sigma \backslash \{x\}$ if $x \in \Sigma$ and Σ otherwise. The introduction of locations determines a change in the behaviour of the free non-linear variables that become nominal: by the separation condition, two free nominal variables with different locations cannot be identified. This corresponds to restricting the application of meta-level contraction — as implicit in the double-entry sequent formulation. Rule *Contr* in the proof system has a more technical character [17] and it is unaffected by the separation condition.

The rule $\hat{\exists} R$ introduces $\lfloor$ on the left, whereas $\hat{\exists} L$ eliminates it. Notice that $\lfloor$ is not treated as standard constructor in the rules — we do allow it to appear in positive position with proper locations. There are no axioms and no right introduction rules for $\lfloor$, and it is not possible to derive a proper location from Γ, as all variables declared in Γ are of closed types. With the given restriction in place, only improper locations can be un-linearised, i.e. $\vdash (!\alpha \lfloor D) \multimap \alpha \lfloor D$ with D closed, and moreover $\vdash (!\forall x.\alpha \lfloor x) \multimap \forall x.\alpha \lfloor x$, but $\nvdash \alpha \lfloor D \multimap \alpha \lfloor D$.

Intuitively, the $\hat{\exists}L$ rule binds a name (a naming variable and a location), extending the schema of the standard existential rule. The $\hat{\exists}R$ rule creates a name and hides it (both naming term and location), replacing exhaustively the term with a bound variable in the type. Notice that locations may occur in negative positions either free, bound (with $\multimap$) or hidden bound (with $\hat{\exists}$), and may occur in positive positions only hidden (with $\hat{\exists}$), whether bound or free. A term is a *location term* when it evaluates to a location. As there is no right introduction of $\lfloor$, we do not need to consider complex location terms explicitly. The operator associated to $\hat{\exists}$ can be defined as

$$\hat{\varepsilon}(D|n).M :: \hat{\exists}x : \beta.\alpha \;=_{df}\; (D :: \beta) \otimes (M :: \alpha[D/x]) \otimes (n :: \beta \lfloor D)$$

for a non-linear term $D :: \beta$, with closed β and x not occurring in D, that additionally satisfies a *freshness condition*: $FV(D) \cap FV(\alpha) = \emptyset$.

The definition of $\hat{\varepsilon}$ is based on that of proof-and-witness pair associated with the interpretation of existential quantifier, in standard λ-calculus [23] as well as in its linear version [4, 17] — however, here a location is added as evidence that the witness is located. The location n is a linear term — this changes the nature of the operator, giving it a resource-bound character.

The freshness condition ensures that the occurrences of the name are the same as the occurrences of the naming term in the main type, and makes the introduction rules of $\hat{\exists}$ essentially invertible, unlike standard existential quantification. The freshness condition is trivially satisfied in the case of $\hat{\exists}L$. In the case of $\hat{\exists}R$, it follows from the fact that $\alpha \to \alpha$ can be derived from Γ_1, x, whereas D can be derived from Γ_2 — assuming that Γ_1 and Γ_2 are disjoint, and that x does not occur in D. Unlike standard linear logic rules, the definition of $\hat{\exists}R$ involves splitting the non-linear context.

The following statements can be proved by induction on the definition of derivation, using the separating condition and linearity of locations. Unlike in double-entry formulations of standard ILL, rule Weakening is explicitly needed here, in order to prove Cut elimination for the $\hat{\exists}$ case.

Prop. 1 (1) Rules *Cut* and *!Cut* can be eliminated without loss for provability.
(2) Given a derivation $[\Sigma]; \Gamma; \Delta \vdash N :: \alpha$
(2.a) it is possible to define a surjective function *Loc* from the free nominal variables in Σ to the set of the naming terms *Names*, such that $Loc(x) = D$ iff $x \in FN(D)$ and $D \in Names$.
(2.b) given a non-linear closed type α such that neither a closed term $D :: \alpha$ nor a term of type $\forall x : \alpha.\alpha \lfloor x$ are derivable from Γ, there is a one-to-one correspondence between the α-locations in negative positions and those (hidden) in positive positions.

Prop. 2 The following formulas are provable

$$\vdash (\hat{\exists}x : \alpha.\beta) \mathrel{\hat{=}} (\hat{\exists}y : \alpha.\beta[y/x]) \qquad (y \text{ not in } \beta)$$
$$\vdash (\hat{\exists}xy : \alpha.\gamma) \mathrel{\hat{=}} (\hat{\exists}yx : \alpha.\gamma)$$
$$\vdash (\hat{\exists}x : \alpha.\beta \otimes \gamma) \mathrel{\hat{=}} (\beta \otimes \hat{\exists}x : \alpha.\gamma) \qquad (x \text{ not in } \beta)$$
$$\vdash (\hat{\exists}x : \alpha.\beta \multimap \gamma) \multimap (\beta \multimap \hat{\exists}x : \alpha.\gamma) \qquad (x \text{ not in } \beta)$$

Notice that in general, an operator ν can be characterised as name restriction when it satisfies the following properties [7].

α-renaming: $\nu y.N \equiv \nu z.N[z/y]$, avoiding variable capture
permutation: $\nu xy.N \equiv \nu yx.N$
scope extrusion: $\nu x.N_1 \otimes N_2 \equiv N_1 \otimes \nu x.N_2$, with x not in N_1
η-congruence: $\nu x.N \equiv N$, with x not in N

By the first three formulas in Prop. 2, $\hat{\exists}$ satisfies properties of α-renaming, exchange and distribution over $\otimes$, and therefore $\hat{\varepsilon}$ satisfies the corresponding properties of restriction. On the other hand, $\hat{\exists}$ does not generally satisfy η-congruence, i.e. it cannot be proved that α is equivalent to $\hat{\exists}x.\ \alpha$ when x does not occur free in α (neither sense of linear implication holds).

It is not difficult to see that the following formulas, which are all valid for existential quantification, fail for $\hat{\exists}$

Prop. 3 (1) $\nvdash (\hat{\exists}x:\beta.\ \alpha(x,x)) \multimap \hat{\exists}xy:\beta.\ \alpha(x,y)$
(2) $\nvdash \forall x:\beta.\ (\hat{\exists}z:\beta.\alpha(z,z)) \multimap \hat{\exists}y:\beta.\alpha(y,x)$
(3) $\nvdash (\hat{\exists}yx:\beta.\ \alpha_1(x)\otimes\alpha_2(x)) \multimap (\hat{\exists}x:\beta.\alpha_1(x))\otimes\hat{\exists}x:\beta.\alpha_2(x)$

In fact, each of the above formulas can be given graphical interpretations that correspond to basic breaches of the DPO conditions [22].

3.1 Proof rules

$$\frac{}{[\emptyset];\Gamma;u::\alpha\vdash u::\alpha \quad \text{with } \alpha \text{ atomic}}\ LId \qquad \frac{}{[\emptyset];\Gamma,x::\alpha;\cdot\vdash x::\alpha \quad \text{with } \alpha \text{ closed}}\ UId$$

$$\frac{\begin{array}{c}[\Sigma_2];\Gamma_2,x::\beta;\cdot\vdash N::\alpha\multimap\alpha \\ [\Sigma_1];\Gamma_1;\cdot\vdash D::\beta \qquad\qquad [\Sigma_1,\Sigma_2];\Gamma_1,\Gamma_2;\Delta\vdash M::\alpha[D/x]\end{array}}{[\Sigma_1,\Sigma_2,FV(D)];\Gamma_1,\Gamma_2;\Delta,n::\beta\,\lfloor D\vdash \hat{\varepsilon}(D|n).M::\hat{\exists}x:\beta.\alpha}\ \hat{\exists}R$$

$$\frac{[\Sigma,z];\Gamma,z::\beta;\Delta,n::\beta\,\lfloor z,v::\alpha\vdash N::\gamma}{[\Sigma];\Gamma;\Delta,u::\hat{\exists}z:\beta.\ \alpha\vdash \mathsf{let}\ \hat{\varepsilon}(z|n).v=u\ \mathsf{in}\ N::\gamma}\ \hat{\exists}L$$

$$\frac{[\Sigma];\Gamma,x::\beta;\Delta\vdash M::\alpha}{[\Sigma-x];\Gamma;\Delta\vdash \lambda x.\ M::\forall x:\beta.\ \alpha}\ \forall R \qquad \frac{[\Sigma];\Gamma;\Delta,u::\alpha\vdash M::\beta}{[\Sigma];\Gamma;\Delta\vdash \hat{\lambda}u:\alpha.\ M::\alpha\multimap\beta}\ \multimap R$$

$$\frac{[\Sigma_1];\Gamma;\cdot\vdash D::\beta \quad [\Sigma_2];\Gamma;\Delta,v::\alpha[D/x]\vdash N::\gamma}{[\Sigma_1,\Sigma_2];\Gamma;\Delta,u::\forall x:\beta.\alpha\vdash \mathsf{let}\ v=uD\ \mathsf{in}\ N::\gamma}\ \forall L$$

$$\frac{[\Sigma_1];\Gamma;\Delta_1\vdash M::\alpha \quad [\Sigma_2];\Gamma;\Delta_2,u::\beta\vdash N::\gamma}{[\Sigma_1,\Sigma_2];\Gamma;\Delta_1,\Delta_2,v::\alpha\multimap\beta\vdash \mathsf{let}\ u=v\hat{}M\ \mathsf{in}\ N::\gamma}\ \multimap L$$

$$\frac{[\Sigma_1];\Gamma;\Delta_1\vdash M::\alpha \quad [\Sigma_2];\Gamma;\Sigma_2;\Delta_2\vdash N::\beta}{[\Sigma_1,\Sigma_2];\Gamma;\Delta_1,\Delta_2\vdash M\otimes N::\alpha\otimes\beta}\ \otimes R \qquad \frac{[\Sigma];\Gamma;\Delta,u::\alpha,v::\beta\vdash N::\gamma}{[\Sigma];\Gamma;\Delta,w::\alpha\otimes\beta\vdash \mathsf{let}\ u\otimes v=w\ \mathsf{in}\ N::\gamma}\ \otimes L$$

$$\frac{}{[\emptyset];\Gamma;\cdot\vdash \mathsf{nil}::\mathbf{1}}\ \mathbf{1}R \qquad \frac{[\Sigma];\Gamma;\Delta\vdash N::\alpha}{[\Sigma];\Gamma;\Delta,u::\mathbf{1}\vdash \mathsf{let}\ \mathsf{nil}=u\ \mathsf{in}\ N::\alpha}\ \mathbf{1}L$$

$$\frac{[\Sigma];\Gamma;\cdot\vdash M::\alpha}{[\Sigma];\Gamma;\cdot\vdash !M::!\alpha}\ !R \qquad \frac{[\Sigma];\Gamma,x::\alpha;\Delta\vdash N::\beta}{[\Sigma];\Gamma;\Delta,u::!\alpha\vdash \mathsf{let}\ !x=u\ \mathsf{in}\ N::\beta}\ !L$$

$$\frac{[\Sigma];\Gamma;\Delta \vdash N :: \alpha}{[\Sigma];\Gamma,\Gamma';\Delta \vdash \mathsf{discard}(\Gamma')\ \mathsf{in}\ N :: \alpha}\ \mathit{Weak} \qquad \frac{[\Sigma];\Gamma,x :: \alpha;\Delta,u :: \alpha \vdash N :: \gamma}{[\Sigma];\Gamma,x :: \alpha;\Delta \vdash \mathsf{let}\ u = \mathsf{copy}(x)\ \mathsf{in}\ N :: \gamma}\ \mathit{Contr}$$

$$\frac{[\Sigma];\Gamma;\Delta \vdash N :: \alpha \quad [\Sigma'];\Gamma;\Delta',u :: \alpha \vdash M :: \beta}{[\Sigma,\Sigma'];\Gamma;\Delta,\Delta' \vdash \mathsf{let}\ u = N\ \mathsf{in}\ M :: \beta}\ \mathit{Cut}$$

$$\frac{[\Sigma];\Gamma;\cdot \vdash D :: \alpha \quad [\Sigma'];\Gamma,x :: \alpha;\Delta \vdash M :: \beta}{[\Sigma,\Sigma'[FV(D)/x]];\Gamma;\Delta[D/x] \vdash \mathsf{let}\ x = D\ \mathsf{in}\ M :: \beta}\ !\mathit{Cut}$$

4 Graphs in HILL

It is possible to embed GT systems in HILL, along lines given in [22] — though there the logic allowed only for variables as naming terms, making it harder to deal with hierarchical graphs. Here instead a node can be represented as non-linear term $D :: T$ where $T = !A$ and A is an atomic closed type, for which we can assume no closed terms are given. This makes it possible to deal with granular representations in which nodes can be subgraphs.

An edge can be represented as a dependently typed function variable $u :: \forall x_1 : T_1,\ldots,x_k : T_k.E(x_1,\ldots,x_k)$. An edge component can be derived as a sequent

$$[\Sigma_1,\ldots,\Sigma_k];\Gamma;u :: \forall x_1 : T_1,\ldots,x_k : T_k.E(x_1,\ldots,x_k) \vdash u\ D_1 \ldots\ D_k :: E(D_1,\ldots,D_k)$$

from the assumptions $[\Sigma_1];\Gamma;\cdot \vdash D_1 :: T_1 \quad \ldots \quad [\Sigma_k];\Gamma;\cdot \vdash D_k :: T_k$.

The same component with hidden node names can be represented as

$$[\Sigma'];\Gamma;n_1 :: T_1 \lfloor D_1,\ldots,n_k :: T_k \lfloor D_k,u :: \forall x_1 : T_1,\ldots,x_k : T_k.E(x_1,\ldots,x_k) \vdash \varepsilon(D_1|n_1)\ldots(D_k|n_k).u\ x_1 \ldots\ x_k :: \hat{\exists} x_1 : T_1,\ldots,x_k : T_k.E(x_1,\ldots,x_k)$$

where $\Sigma' = [\Sigma_1,FV(D_1),\ldots,\Sigma_k,FV(D_k)]$. The empty graph can be represented as $[\emptyset];\Gamma;\cdot \vdash \mathsf{nil} :: \mathbf{1}$. The parallel composition of two components $[\Sigma_1];\Gamma;\Delta_1 \vdash G_1 :: \gamma_1$ and $[\Sigma_2];\Gamma;\Delta_2 \vdash G_2 :: \gamma_2$ can be represented as

$$[\Sigma_1,\Sigma_2];\Gamma;\Delta_1,\Delta_2 \vdash G_1 \otimes G_2 :: \gamma_1 \otimes \gamma_2$$

As a further example, assuming $[\Sigma];\Gamma;\cdot \vdash D :: T$ an isolated node can be represented as

$$[\Sigma,FV(D)];\Gamma;n :: T \lfloor D \vdash \varepsilon(D|n).\mathsf{nil} :: \hat{\exists} x : T.\mathbf{1}$$

It is not difficult to see how an encoding of hypergraphs into HILL can be defined inductively along these lines. Let G be a typed hypergraph, and let it be closed (i.e. without external nodes). We can define a graph signature $\langle \Delta_G^N, \Delta_G^E \rangle$, where Δ_G^N are the locations that represent the nodes of G, and Δ_G^E are the linear variables that represent the edges of G. We call *graph formulas* those in the $\mathbf{1}, \otimes, \hat{\exists}, \forall, \lfloor$ fragment of the logic containing as primitive types only node and edge types, such that quantification ranges on node types only. We say that a graph formula γ is in normal form whenever $\gamma = \hat{\exists}(\overline{x : T}).\ \alpha$, where either $\alpha = \mathbf{1}$ or $\alpha = E_1(\overline{x}_1) \otimes \ldots \otimes E_k(\overline{x}_k)$, with $\overline{x :: T}$ a sequence of typed variables. The formula is closed if $\overline{x}_i \subseteq \overline{x}$ for each $1 \leq i \leq k$. G can be represented by a derivation

$$[FN(\Delta_G^N)];\Gamma;\Delta_G^N,\Delta_G^E \vdash N_G :: \gamma$$

where γ is a closed normal graph formula that we call *representative* of G. This encoding can be extended to an abstract hypergraphs $I \to G$, by representing edges with linear variables Δ_G^E and internal nodes with locations Δ_G^N as before, and by representing interface nodes as free variables that can be λ-abstracted. The representative γ has then form $\forall x_1 : T_1, \ldots, x_j : T_j.\gamma'$, where γ' is a normal graph formula, and $x_1 : T_1, \ldots, x_j : T_j$ are the open nodes. This translation generalises that given in [22].

4.1 Transformation rules

Graph transformation can be represented by linear inference. In particular, a direct transformation $G \Longrightarrow H$, where G, H are closed hypergraphs, can be encoded logically as $\gamma_G \multimap \gamma_H$, where γ_G, γ_H are representatives of G and H, respectively. Let $\pi(p) = \Lambda K.L \Longrightarrow R$ be a DPO transformation rule with discrete interface, i.e. such that K is the set of the typed nodes that are shared between L and R, and such that none of them is isolated in both L and R. Then p can be represented logically as non-linear term

$$z_p ::! \forall x_1 : T_1, \ldots, x_k : T_k.\gamma_L \multimap \gamma_R$$

where γ_L, γ_R are normal graph formulas, representatives of L and R respectively, and $x_1 : T_1, \ldots, x_k : T_k$ represent the nodes in K. The ! closure guarantees unrestricted applicability, universally quantified variables represent the rule interface, and linear implication represents transformation.

As shown in the double-pushout diagram (section 2), the application of rule $\pi(p)$ determined by morphism m to a closed hypergraph G, resulting in a closed hypergraph H, can be represented up to isomorphism as a derivation of an H representative $\alpha_H = \hat{\exists}\overline{y : T_y}.\beta_H$ from a G representative $\alpha_G = \hat{\exists}\overline{y : T_y}.\beta_G$, based on z_p and on the multiple substitution $[\overline{z : T_z} \xleftarrow{d} \overline{x : T_x}]$ of the free variables in γ_L, γ_R, corresponding to the interface morphism d (not required to be injective) in the diagram. A transformation determined by an application of the rule can be proved correct, up to isomorphism, by the fact that the following is a derivable rule

$$\frac{\begin{array}{ll}[\emptyset];\Gamma;\cdot \vdash \alpha_G \hat{=} \alpha_{G'} & \alpha_{G'} = \hat{\exists}\overline{y : T_y}.\alpha_L[\overline{z : T_z} \xleftarrow{d} \overline{x : T_x}] \otimes \alpha_C \\ [\emptyset];\Gamma;\cdot \vdash \alpha_H \hat{=} \alpha_{H'} & \alpha_{H'} = \hat{\exists}\overline{y : T_y}.\alpha_R[\overline{z : T_z} \xleftarrow{d} \overline{x : T_x}] \otimes \alpha_C\end{array}}{[\emptyset];\Gamma;\forall\overline{x : T_x}.\alpha_L \ \multimap \ \alpha_R \vdash \alpha_G \multimap \alpha_H} \ \stackrel{p,m}{\Longrightarrow}$$

where $\overline{z : T_z} \subseteq \overline{y : T_y}$, as G and H are closed.

Prop. 4 The application of a transformation rule to a closed graph representative implies linearly a closed graph representative that is determined up to graph isomorphism by the instantiation of the rule interface variables (morphism d). The match determined by d (up to isomorphism) satisfies the gluing condition on both sides — with respect to the rule instance premise and the initial graph, and with respect to the rule instance consequence and the resulting graph — and therefore satisfies the DPO conditions (Proof: since $\hat{\exists}$ behaves injectively with respect to multiple instantiations, as from Prop. 1(2.b), and satisfies the properties of restriction in Prop. 2).

As to reachability, a sequent $\Gamma; P_1, \ldots, P_k, G_0 \vdash G_1$, where Γ does not contain any rule, can express that graph G_1 is reachable from the initial graph G_0 by applying rules $P_1 = \forall\overline{x}_1.\alpha_1 \multimap \beta_1, \ \ldots, \ P_k = \forall\overline{x}_k.\alpha_k \multimap \beta_k$ once each, abstracting from the application order. A sequent $\Gamma, P_1, \ldots, P_k; G_0 \vdash G_1$ can express that G_1 is reachable from G_0 by the same rules, regardless of whether or how many times they

are applied. The parallel application of rules $\forall \overline{x}_1.\alpha_1 \multimap \beta_1, \forall \overline{x}_2.\alpha_2 \multimap \beta_2$ can be represented as application of $\forall \overline{x}_1\overline{x}_2.(\alpha_1 \multimap \beta_1) \otimes (\alpha_2 \multimap \beta_2)$, as distinct from $\forall \overline{x}_1\overline{x}_2.\alpha_1 \otimes \alpha_2 \multimap \beta_1 \otimes \beta_2$.

4.2 Example

We give an example of logic derivation that represents the application of a transformation rule (graphically represented in Fig. 1), conveniently simplifying the notation, by making appropriate naming choices.

$$
\dfrac{
\dfrac{
\begin{array}{l}
\dfrac{
\begin{array}{l}
\Gamma^*; A(x_1,x_2) \vdash A(x_1,x_2) \\
\Gamma^*; B(x_2) \vdash B(x_2) \\
\Gamma^*; C(x_1) \vdash C(x_1) \\
\Gamma^*; D(x_5,x_6) \vdash D(x_5,x_6)
\end{array}
}{
\begin{array}{l}
\Gamma^*; A(x_1,x_2), B(x_2), C(x_1), D(x_5,x_6) \vdash \\
\quad C(x_1) \otimes A(x_1,x_2) \otimes D(x_5,x_6) \otimes B(x_2)
\end{array}
} \otimes R^* \\
\Gamma^* \vdash x_1 \qquad \Gamma^* \vdash x_2 \\
\Gamma^* \vdash x_5 \qquad \Gamma^* \vdash x_6
\end{array}
}{
\begin{array}{l}
\Gamma^*; n_1 \lfloor x_1, n_2 \lfloor x_2, n_5 \lfloor x_5, n_6 \lfloor x_6, \\
A(x_1,x_2), B(x_2), C(x_1), D(x_5,x_6) \vdash \gamma_H \\
\quad \Gamma^* = \Gamma, x_5, x_6
\end{array}
} \hat{\exists} R^*
}{
\begin{array}{l}
\Gamma; n_1 \lfloor x_1, n_2 \lfloor x_2, A(x_1,x_2), B(x_2), \\
\hat{\exists} y_3, y_4 : \alpha_3.C(x_1) \otimes D(y_3,y_4) \vdash \gamma_H
\end{array}
} \otimes L, \hat{\exists} L^*
$$

$$\vdots$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\begin{array}{l}
\dfrac{
\begin{array}{l}
\Gamma; A(x_1,x_3) \vdash A(x_1,x_3) \\
\Gamma; C(x_1) \vdash C(x_1)
\end{array}
}{
\begin{array}{l}
\Gamma; C(x_1), A(x_1,x_3) \\
\quad \vdash C(x_1) \otimes A(x_1,x_3)
\end{array}
} \otimes R \\
\Gamma \vdash x_3
\end{array}
}{
\begin{array}{l}
\Gamma; n_3 \lfloor x_3, C(x_1), A(x_1,x_3) \\
\quad \vdash \hat{\exists} y_2 : \alpha_2.C(x_1) \otimes A(x_1,y_2)
\end{array}
} \hat{\exists} R
\quad
\begin{array}{l}
\vdots \\
\Gamma; n_1 \lfloor x_1, n_2 \lfloor x_2, \\
A(x_1,x_2), B(x_2), \\
\hat{\exists} y_3, y_4 : \alpha_3.C(x_1) \\
\otimes D(y_3,y_4) \vdash \gamma_H
\end{array}
}{
\begin{array}{l}
\Gamma; n_1 \lfloor x_1, n_2 \lfloor x_2, n_3 \lfloor x_3, C(x_1), A(x_1,x_2), A(x_1,x_3), B(x_2), \\
(\hat{\exists} y_2 : \alpha_2.C(x_1) \otimes A(x_1,y_2)) \multimap (\hat{\exists} y_3, y_4 : \alpha_3.C(x_1) \otimes D(y_3,y_4)) \vdash \gamma_H
\end{array}
} \multimap L
}{
\begin{array}{l}
\Gamma; n_1 \lfloor x_1, n_2 \lfloor x_2, n_3 \lfloor x_3, C(x_1), A(x_1,x_2), A(x_1,x_3), B(x_2), \delta \vdash \gamma_H \\
\quad \Gamma = \Gamma', x_1, x_2, x_3
\end{array}
} \forall L
}{
\dfrac{\Gamma'; \delta, \gamma_G \vdash \gamma_H}{\Gamma'; \delta \vdash \gamma_G \multimap \gamma_H} \multimap R
} \hat{\exists} L^*
$$

where graphs G, H and rule $\pi(p)$ be represented as follows

$\gamma_G = \hat{\exists} x_1 : \alpha_1, x_2 : \alpha_2, x_3 : \alpha_3.C(x_1) \otimes A(x_1,x_2) \otimes A(x_1,x_3) \otimes B(x_2)$

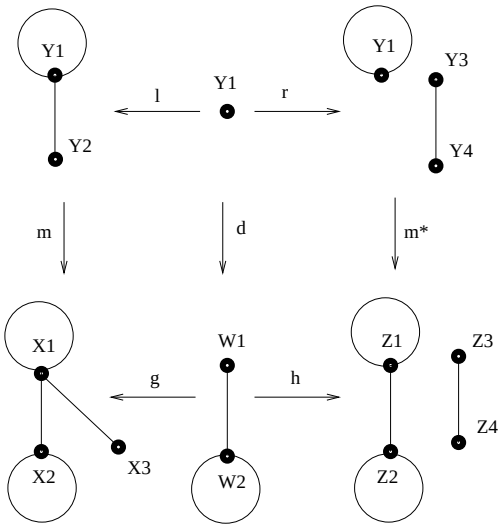

Figure 1: Transformation example

$$\delta = \forall y_1 : \alpha_1.(\hat{\exists} y_2 : \alpha_2.C(y_1) \otimes A(y_1, y_2)) \multimap (\hat{\exists} y_3, y_4 : \alpha_3.C(y_1) \otimes D(y_3, y_4)$$
$$\gamma_H = \hat{\exists} z_1 : \alpha_1, z_2 : \alpha_2, z_3 z_4 : \alpha_3.C(z_1) \otimes A(z_1, z_2) \otimes D(z_3, z_4) \otimes B(z_2)$$

The derivation shows that the graph represented as γ_H can be obtained by a single application to the graph represented as γ_G of the rule represented as δ . The transformation can be represented logically as sequent $\Gamma'; \delta \vdash \gamma_G \multimap \gamma_H$, easily provable by backward application of the proof rules, as shown. The fact that naming terms here are variables makes the book-keeping of free nominal variables straightforward (and annotation unnecessary).

5 Conclusion and further work

We have discussed how to represent DPO-GTS in a quantified extension of ILL, to reason about concurrency and reachability at the abstract level. We focussed on abstraction from name identity, an aspect that in hyperedge replacement formulations of GT is often associated with name restriction [7]. We used an approach that, with respect to nominal logic, appears comparatively closer to [21] than to [20] — though our resource-bound quantifier is essentially based on existential quantification, and unlike freshness quantifiers does not seem to be so easily understood in terms of *for all*.

We have followed the general lines of the encoding presented in [22], but we have relied on a more expressive logic, allowing for the use of complex terms as names. With this extension, it becomes possible to go beyond flat hypergraphs as defined in section 2, and to consider structured ones [11, 3]. Moreover, it should be possible to deal with transformation rules that are not discrete, i.e. that include edge components in the interface, by shifting to a representation in which hyperedges, too, are treated as names. However, if such extensions do not appear particularly problematic from the point of view of soundness, they may make completeness results rather more difficult.

References

[1] S. Abramsky (1993): *Computational interpretation of linear logic*. *Theoretical Computer Science* 111.

[2] N. Benton, G. Bierman, V. de Paiva & M. Hyland (1993): *Linear lambda-calculus and categorical models revisited*. In: E. Börger, G. Jäger, Kleine H. Büning, S. Martini & M. Richter, editors: *Proceedings of the Sixth Workshop on Computer Science Logic*. Springer Verlag, pp. 61–84.

[3] Giorgio Busatto, Hans-Jörg Kreowski & Sabine Kuske (2005): *Abstract hierarchical graph transformation*. *Mathematical Structures in Computer Science* 15(4), pp. 773–819.

[4] I. Cervesato & F. Pfenning (2002): *A linear logical framework*. *Information and Computation* 179(1), pp. 19–75.

[5] Iliano Cervesato & Andre Scedrov (2006): *Relating State-Based and Process-Based Concurrency through Linear Logic*. *Electron. Notes Theor. Comput. Sci.* 165, pp. 145–176.

[6] David Clarke (2007): *Coordination: Reo, nets, and logic*. In: *FMCO 2007, LNCS* 5382. pp. 226–256.

[7] Andrea Corradini, Ugo Montanari & Francesca Rossi (1994): *An abstract machine for concurrent modular systems: CHARM*. *Theoretical Computer Science* 122, pp. 165–200.

[8] B. Courcelle (1997): *The expression of graph properties and graph transformation in monadic second-order logic*. In: G. Rozenberg, editor: *Handbook of Graph Grammars and Computing by Graph Transformation*, 1. World Scientific, pp. 313–400.

[9] Lucas Dixon, Alan Smaill & Alan Bundy (2006): *Planning as Deductive Synthesis in Intuitionistic Linear Logic*. Technical Report, University of Edinburgh.

[10] Mike Dodds & Detlef Plump (2008): *From hyperedge relpacement to separation logic and back*. In: *ICGT 2008 — Doctoral Symposium*.

[11] Frank Drewes, Berthold Hoffmann & Detlef Plump (2002): *Hierarchical graph transformation*. *J. Comput. Syst. Sci.* 64(2), pp. 249–283.

[12] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of algebraic graph transformation*. Springer.

[13] Murdoch J. Gabbay & J. Cheney (2004): *A Sequent Calculus for Nominal Logic*. In: *19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*. pp. 139–148.

[14] Dan Hirsch & Ugo Montanari (1999): *Consistent transformations for software architecture styles of distributed systems*. *Electr. Notes Theor. Comput. Sci.* 28.

[15] D. Miller (1992): *The pi-calculus as a theory in linear logic: preliminary results*. In: *Workshop on Extensions of Logic Programming*, number 660 in LNCS. Springer, pp. 242–264.

[16] Frank Pfenning (1994): *Structural Cut Elimination in Linear Logic*. Technical Report, Carnagie Mellon University.

[17] Frank Pfenning (2002): *Linear Logic — 2002 Draft*. Technical Report, Carnagie Mellon University.

[18] Andrew Pitts & Ian Stark (1993): *Observable Properties of Higher Order Functions that Dinamycally Create Local Names, or: What's new?* In: *MFCS'93*. Springer, pp. 122–141.

[19] Andrew M. Pitts (2001): *Nominal Logic: A First Order Theory of Names and Binding*. In: *TACS '01: Proc. 4th Int. Symp. on Theoretical Aspects of Computer Software*. Springer, pp. 219–242.

[20] D. J. Pym (2002): *The semantics and proof-theory of the logics of bunched implications*. Applied Logic Series. Kluwer.

[21] Ulrich Schoepp & Ian Stark (2004): *A Dependent Type Theory with Names and Binding*. In: *Computer Science Logic '04*. Springer, pp. 235–249.

[22] Paolo Torrini & Reiko Heckel (2009): *Towards an embedding of Graph Transformation in Intuitionistic Linear Logic*. *CoRR* abs/0911.5525.

[23] A. S. Troelstra & H. Schwitchtenberg (2000): *Basic Proof Theory*. Cambridge University Press.